CS 342 - OPERATING SYSTEMS

PROJECT 4

ABDULLAH CAN ALPAY 21702686 - SECTION 3

ÜMİT YİĞİT BAŞARAN 21704103 - SECTION 1

16.05.2021

Approach

As it stated in the specifications, we first generate 2^m bytes of memory file. After that we divide the size into 4KB to get the blocks in the file system. In the first block (block id: 0) we store the size of the file; from the second to fifth block (block ids: 1,2,3,4) we store the bitmaps. In each block we have 4KB bytes, if we convert it to the bits we get 2^{15} bits. That means bitmap can hold $4x2^{15}$ bits (4 is the block number, 2^{15} is the size of each block). Then we used the following 4 blocks to directory names and their corresponding inodes. We allocated 128 bytes (size of FAT) for each name and inode. Thus, we can store 32 directory names in one block (4KB / 128), and total 128 directory names (32 * 4). After, we allocated 4 blocks for the File Control Block. In each block we stored 32 block_index (128 bytes is allocated per FAT), it's availability and its size. The index_block is used for showing the blocks used for the specified directory name. Each index_block initially stores -1 (as ffff, since integer is 4 bytes and the block size is 4KB, we can have 1024 integers in index_block). For the remaining blocks, we store the information.

In the File Control Blocks (FCB) (block id: 9, 10, 11, 12) we observed that the system puts 1 in the size while creating the memory file. At first we thought that it would cause problems. However, while creating a file (by calling sfs_create), we overwrite the FCB blocks. We change the corresponding FAT in the FCB.

structs

Open File Entry

```
struct open_file_entry {
    int used;
    char name[110];
    int inode;
    int mode;
    int offset;
    int size;
    int block_count;
};
```

used: Checks whether this entry is used or not
name[110]: Stores the directory name
inode: Stores the inode for quick access
mode: Shows whether it's on READ mode APPEND mode
offset: Stores the last location on the block (0<= offset<=BLOCKSIZE (4096)).
    offset is set 0 if entry opened in READ mode
    offset is set size%4096 if entry opened in APPEND mode.

size: Stores the initial size from FCB when the entry is saved.
block_count: Initially 0. When the entry is opened in READ mode, block_count helps us to know the offset's location.

Index Block

```
struct index_block {
   int index_arr[1024];
};
```

Helped us to find the location in blocks.

File Control Block

```
struct file_control_block {
   int available;
   int index_block;
   int size;
};
```

available: Is set to 1, if the location is used.
index_block: Stores the index_block id.
size: When closing the file, we updated the size and stored it in the FCB. When the directory file is opened, we get the size from FCB.

Directory Item

```
struct directory_item {
   char name[110];
   int inode;
};
```

name: Stores the directory file name
inode: Stores the index in the FCB.

Functions

sfs_read

In sfs_read we need to know where we stopped at. block_count stores fully finished blocks in the directory file. Thus, at the beginning we shift our beginning index according to block_size. This means, we pass integers (that are not -1) in the index block according to block_count. After we found which block to start reading, we added an offset to find the exact location to start at which byte to start reading. Then, we memcpy the blocks and the remaining fraction in the block.

sfs_append

Our biggest challenge is to prevent internal fragmentation in the blocks. We found that internal fragmentation occurs whenever we close and open the directory file. Our offset is set to 0 in sfs_open, thus, no matter what we started from the next block, rather than filling the available block. To prevent this, we add size in the FCB. When we open the directory file, we retrieve the size and with size%4096 we can find where we stopped in the block. After that we start appending from the offset. With that we eliminate the fragmentation problem. Since we add blocks respectively, our last block will be the one that is before the -1 index.

Experiments

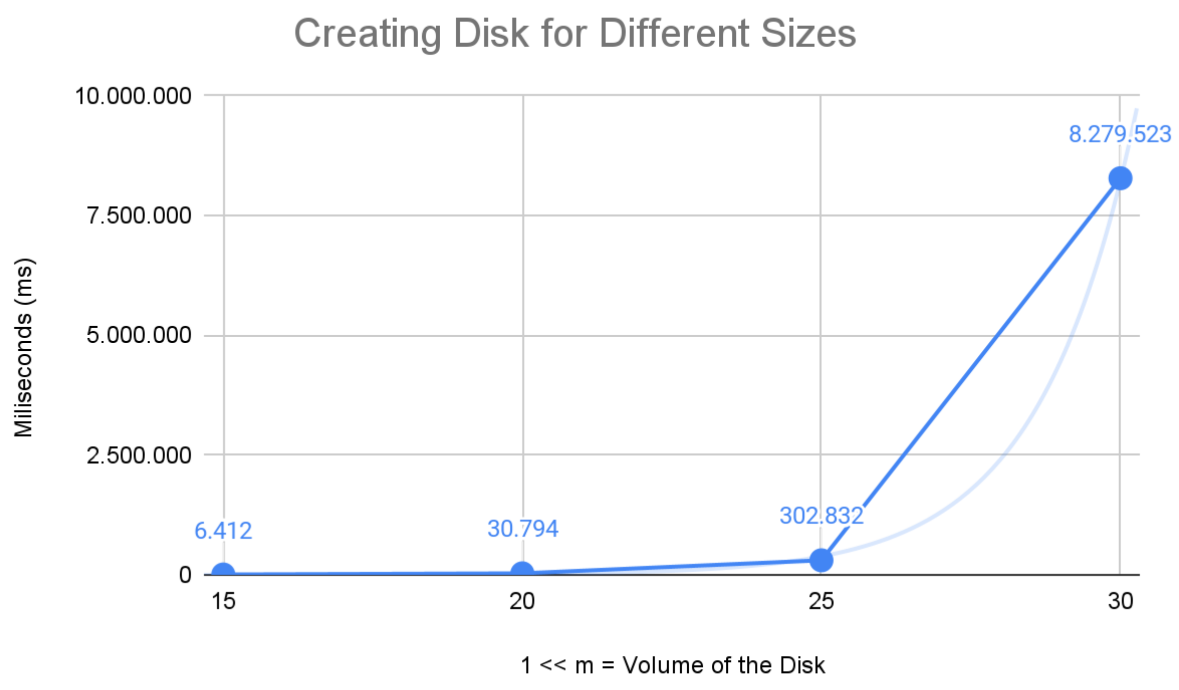

Figure 1: Create disk with different disk volumes

From figure 1, it can be seen that when the m value increases the time of the creating disk increases exponentially. Also, the trendline shows the exponentiality.

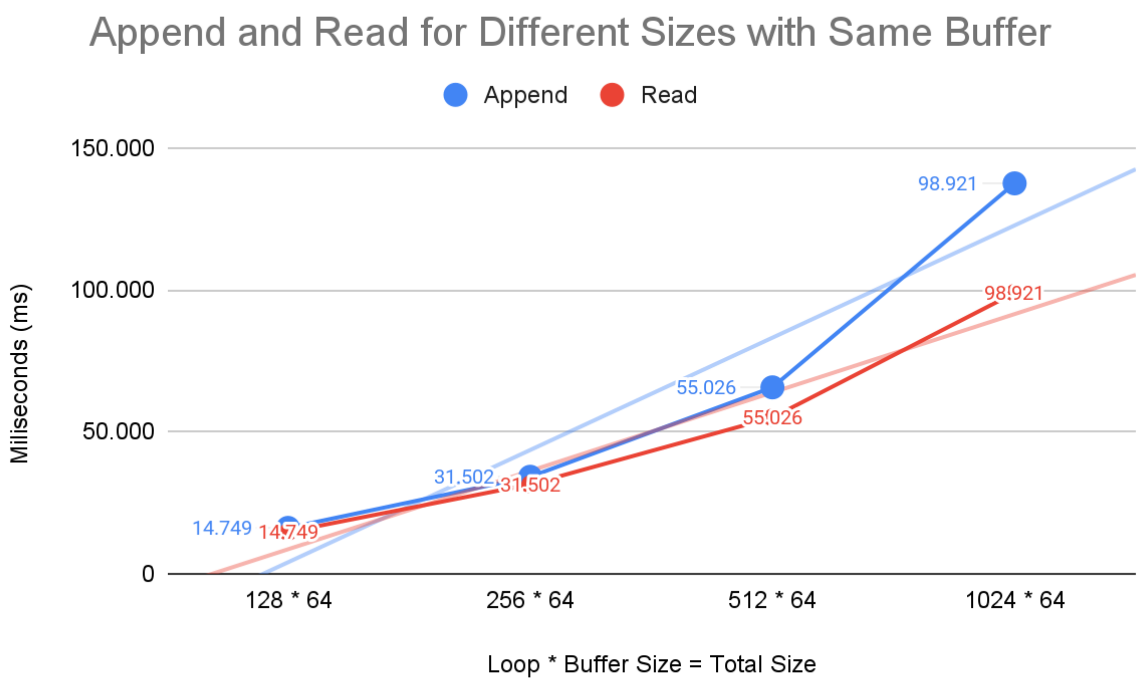

Figure 2 - Append and Read operations - same buffer sizes - different loop values

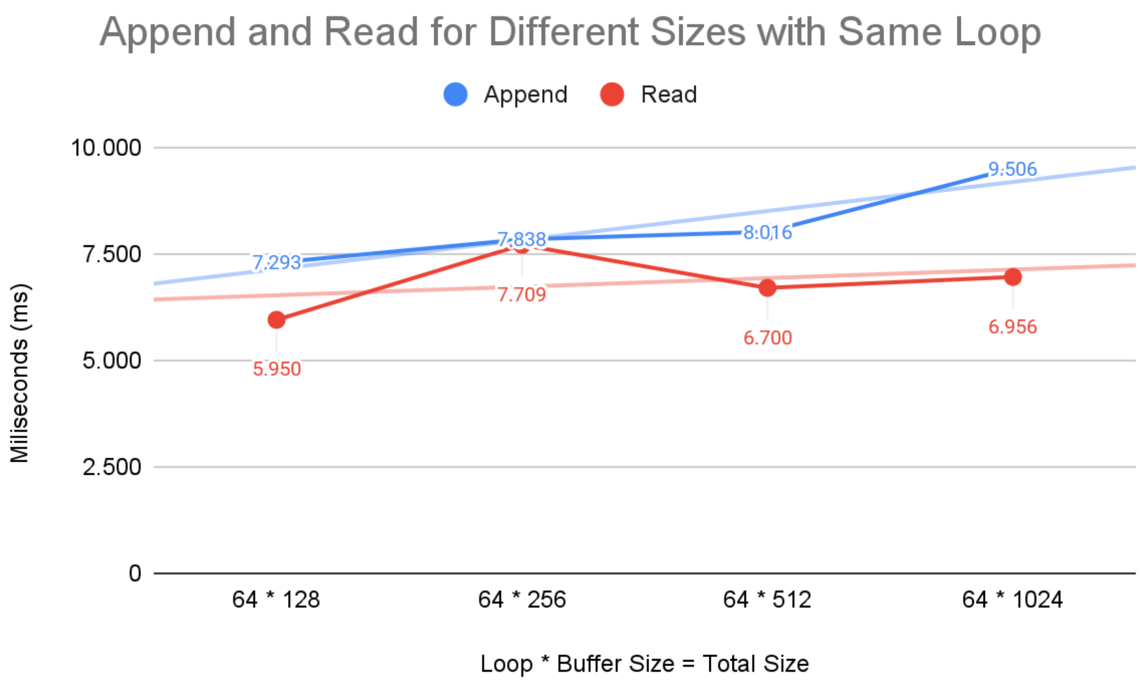

Figure 3 - Append and Read operations - different buffer sizes - same loop values

In the experiment conducted for figure 2, the buffer size is fixed to 64 byte for both read and write buffer. Then, inside different loops this buffer writes and reads data to the disk with different repeat values. From this figure, it can be seen that when the repeat time increases the time also increases linear. By looking at the trendlines of the figure 2, the slope of the “append” trendline is more than the slope of the “read” trendline. So that, when the repeat value increases the difference between append time and read time increases.

In the experiment conducted for figure 3, the repeat size of the write and read loops is fixed however the buffer size of the write and read buffers are increased while conducting experiment. By looking at the figure 3, although the buffer size increases linear the time of the operations change a little. Also, in this graph, the append operation takes more time than read operation, too.

If we compare these two figures, we could say that increasing buffer size is more time efficient than increasing repeat size.